CSRD Rpt. No. 1476

A NEW-GENERATION PARALLELIZING COMPILER SYSTEM

Final R & D Report
ARPA Contract **DABT63-92-C-0033**

David A. Padua
Rudolf Eigenmann
Jay Hoeflinger

September 1996

Center for Supercomputing Research and Development
Coordinated Science Lagoratory
University of Illinois at Urbana-cHampaign
415 Computer and Systems Research Laboratory
1308 West Springfield Avenue
Urbana, IL 61801
Phone: (217) 333-6223

1 Introduction

Parallel computing is an enabling technology for many areas of science and engineering. Parallel computing is also our best investment to continue achieving performance gains as the limits of semiconductor technology are approached. Whereas the design of parallel computers is reasonably well-understood, the programming of these machines remains in its infancy. As a consequence, programming parallel computers is substantially more difficult than programming conventional uniprocessors. In fact, it is commonly said that the development of effective parallel programming techniques is the main challenge faced by high-performance computing today.

Unless this challenge is met, the acceptance of parallel computers will remain slow, thereby hampering progress in computational science and engineering. An appealing strategy to meet this challenge is to use compilers to translate conventional programs into parallel form. Such compilers would enable the execution of existing programs on new parallel machines, thus allowing a seamless transition into parallel computing. With the support of such compilers, not only would legacy codes could be easily ported, but also new programs could be developed in the familiar sequential paradigm, thus liberating the programmer from the complexities of explicit, machine-oriented parallel programming.

This is the final report of the ARPA project entitled *A New-Generation Parallelizing Compiler System* whose objectives were to advance the state of the art in automatic program parallelization and to demonstrate that, with advanced parallelization techniques, real conventional Fortran programs can be automatically translated into effective parallel codes. As can be seen from the discussion below, we have succeeded on both objectives. In fact, the project reported here was the first compiler research project in more than a decade to show substantial progress in automatic parallelization.

As part of this project, we built Polaris, an experimental translator of conventional Fortran programs which targets shared-memory multiprocessors such as the SGI Challenge and the SUN Enterprise server, as well as scalable machines with a global address space, such as the Cray T3D. In Section 2, we briefly describe the internal organization of Polaris. In Section 3, we enumerate the techniques applied by Polaris and elaborate on how they were chosen for implementation. In Section 4, we present an evaluation of the effectiveness of Polaris. and Section 5 covers ongoing work as well as possible future directions of the Polaris project. Finally, in Section 6 we discuss the educational impact of the project as well as its impact on other research groups and on industry. More extensive surveys of the results of this project can be found in [7], [8], [17], and [1]. The references at the end of this document are the publications of the project including theses and technical reports.

2 The Polaris System

Polaris was developed as a vehicle to experiment with program analysis and restructuring techniques and to help in their evaluation by generating executable code for parallel computers with a global address space. We selected these machines as targets of this project, rather than machines requiring message passing, to avoid the complexities of message generation. In this way we were able to focus all our energies on improving the accuracy of the analysis techniques to detect parallelism and on transformation techniques to enable the exploitation of parallelism.

In retrospect, this was clearly the right choice not only because we avoided the distraction of generating messages, but also and as we expected because global address space has become the dominant machine organization. In fact, industry is now rapidly converging towards the global address space model.

Polaris presently accepts Fortran programs and generates parallel Fortran code for the SGI and Sun multiprocessor using vendor-specific directives. Polaris also generates parallel code for the GUIDE compiler from Kuck and Associates Inc., a portable compiler with implementations for most shared-memory multiprocessor machines. We are now working on code generation modules for the Convex Exemplar and the Cray T3D.

We started developing Polaris in March 1993. The system was developed in C++ and presently it contains about 200,000 lines of code. In the design of the internal organization of Polaris, our main objective was to develop a system that would facilitate the development and debugging of the analysis and transformation routines. We chose to build the system around a class hierarchy that represents a Fortran program internally to the translator. The methods in the hierarchy were designed in a such a way that the Fortran program is always well-formed and the associated information, such as the symbol table and the control flow graph, is automatically kept consistent with the program being transformed.

The class hierarchy also includes a number of high-level operations useful to manipulate Fortran programs such as the operations to construct and manipulate sets. A detailed description of the class hierarchy can be found in [12] and [13]. As part of the internal organization, we have also designed a language extension to C++ that we call FORBOL [36]. This is a collection of high-level operations for pattern matching and replacement that allows a very compact expression of a class of frequently applied pattern-based transformations.

The internal organization of Polaris has proven quite effective. Thanks to the object-oriented approach we followed, the number of errors associated with bookkeeping operations decreased substantially relative to the number of errors in a more conventional, non object-oriented, approach. Furthermore, the implementation of transformations was simplified dramatically by the availability of an extensive collection of powerful high-level operations. As a consequence, we were able to complete the implementation of the system in a relatively short time. In fact, the first version of Polaris was completed in only two years with the participation of only six half-time students and two software engineers.

3 Techniques Implemented in Polaris

The most important techniques implemented in Polaris were identified as important in a study of the effectiveness of commercial Fortran parallelizers [11] that preceded this project. In that study, the Perfect Benchmarks, a collection of conventional Fortran programs representing the typical workload of high-performance computers, were compiled for the Alliant FX/80, an eight-processor multiprocessor popular in the late 1980s. For each program, we computed the speedup as a measure of the quality of the parallelization. Although the speedups obtained were satisfactory in a few cases, the Alliant Fortran compiler failed to deliver any significant speedup for the majority of the programs.

The main reason for this failure was the inability of the Alliant compiler to analyze and transform into parallel form some of the most time-consuming multiply-nested loops in the Perfect Benchmarks. In retrospect, this was not surprising because the parallelization module

of the Alliant compiler was originally developed for vectorization. Commercial multiprocessors, such as the Alliant FX/80, were relatively new in the late 1980s, and the parallelization modules of their compilers were usually based on translators for vector machines. Vectorizers, in their quest for isolating loop kernels and transforming them into vector operations, focus primarily on innermost loops. Compilers for multiprocessors, in contrast, must focus on parallelizing outermost loops in order to reduce the effect of the overhead associated with parallel execution. Furthermore, with outer loop parallelization it is possible to take advantage of the true power of multiprocessing and in this way increase the fraction of code executed in parallel.

Our study showed that extending the four most important analysis and transformation techniques traditionally used for vectorization - dependence analysis, privatization, induction variable substitution, and reduction substitution - led to significant increases in speedup. The bulk of our energies went to the development and implementation of the enhanced version of these four techniques plus two others needed for interprocedural analysis: *Inlining* and *Interprocedural Value Propagation.*

In the area of dependence analysis, we developed a new technique, the *Range Test*, capable of dealing with non-affine subscript expressions. The implementation of this test was supported by powerful computer algebra algorithms and propagation techniques [3, 6, 2, 4, 9, 5] . For privatization, we developed a new strategy based on flow analysis and also supported by powerful computer algebra and attribute propagation algorithms [33, 35, 34, 32]. New techniques were also developed for induction and reduction substitution [15, 19, 20, 21].

We also implemented a very powerful and robust inlining module to help, albeit in a limited way, with interprocedural analysis [14]. Polaris inlines automatically all subroutines which meet a set of criteria. We call this last strategy *Autoinlining.* The other technique for interprocedural analysis applied by Polaris is the Interprocedural Value Propagation (IPVP) of integer variables. IPVP clones subroutines when different call sites supply different values to the parameters. As can be seen below, autoinlining and IPVP, although not as powerful as a comprehensive interprocedural analysis algorithm, improved the accuracy of program analysis in several cases.

4 Effectiveness of Polaris Techniques

Figure 1 presents a speedup comparison between Polaris and SGI's parallelizer PFA for sixteen benchmark programs, from three different sources:

- `arc2d`, `bdna`, `flo52`, `mdg`, `ocean`, and `trfd` from the Perfect Benchmarks suite;
- `applu`, `appsp`, `hydro2d`, `su2cor`, `swim`, `tfft2`, `tomcatv`, and `wave` from the SPEC95 benchmark suite; and,
- `cmhog` and `cloud3d` from NCSA.

The first two suites are well-known. The third source is a collection of programs currently being used in research at the National Center for Supercomputing Applications (NCSA). These programs are of moderate size, containing between 10,000 and 20,000 lines each.

The programs were executed (in real-time mode for timing accuracy) on an eight processor SGI Challenge with 150 MHz R4400 processors. Figure 1 shows that Polaris delivers, in many cases, substantially better speedups than PFA.

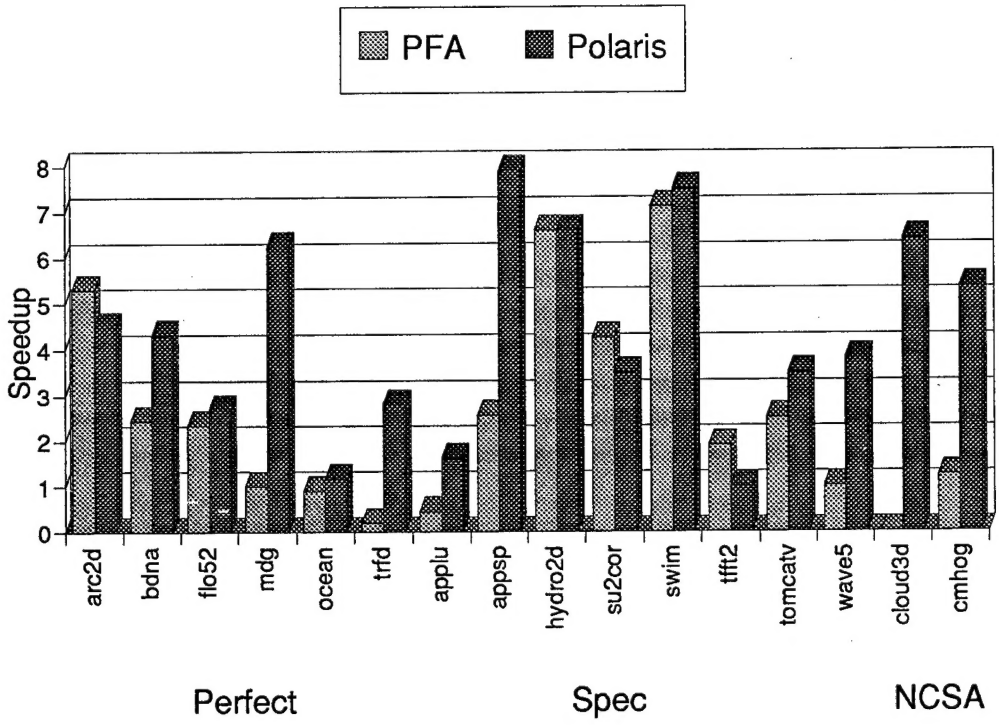

Figure 1: Speedup Comparison Between the SGI PFA Compiler and Polaris

In three of the sixteen programs, PFA produces better speedups than Polaris. The reason is that PFA uses an elaborate code generation strategy that includes loop transformations, such as loop interchanging, unrolling, and fusion, which, when applied to the right loops, improve performance by decreasing overhead, enhancing locality, and facilitating the detection of instruction-level parallelism. However, the elaborate strategy has a negative effect for the program `tomcatv`, in that PFA detects as much parallelism as does Polaris, but the Polaris-transformed program runs faster.

To evaluate the effectiveness of the six techniques implemented in Polaris, we transformed each program six times, with one technique either turned off or weakened in each, and then compared those with the program compiled with all techniques at full strength. In the case of Autoinlining and Interprocedural Value Propagation, the techniques were turned off completely. In the case of Privatization, Dependence Analysis, Induction Variable Recognition, and Reduction Recognition, only the advanced components of the techniques were turned off. For the Dependence Analysis measurement, the Equality and GCD Tests were applied, while the Range Test was not. For the Reduction Recognition measurements, only scalar reduction variables were considered. For Privatization, only scalar privatization was applied. For the Induction Recognition measurement, only induction variables with linear closed forms were taken into account.

Figure 2 presents the results of the six experiments conducted for each code. The height of the bar at (P, T) represents, in logarithmic scale, the percentage of the total number of loops in program P which become serialized by disabling technique T. From Figure 2, it can be seen that all techniques enable parallelization of loops from many programs in the collection. This shows that although the inspiration for the techniques came from analyzing programs in only the Perfect Benchmarks, they are also effective on programs outside the Perfect Benchmarks.

5 The Future: Detecting the Remaining Parallelism and Improving Performance

Polaris has detected much, but not all, of the parallelism available in our set of benchmark codes. A careful analysis of the loops in our benchmark codes which Polaris could parallelize, but does not, shows several areas where improvement is possible.

First, we need to extend all of our analysis interprocedurally. Although IPVP has proven quite useful, a true interprocedural framework for all analysis is still needed. However, due to the large amount of information required by our analysis algorithms, a traditional global propagation algorithm would be too inefficient. We are, therefore, focusing on a demand-driven strategy that would allow high accuracy without significantly increasing the analysis time.

Second, improving our analysis techniques for dependence and privatization is important. When compile time analysis fails, analysis code needs to be generated for use at runtime. This will allow Polaris to parallelize even loops with access patterns determined by input values.

In fact, although not used to obtain the results described in the preceding section, we have also developed and implemented a novel *runtime dependence and privatization test* [26, 28, 31, 30, 27, 25, 23, 29, 24, 22, 16] to enable the parallelization of those codes whose memory access pattern is not known at compile time. Such situations arise, for example, in sparse computations. Preliminary results look quite promising. We plan to conduct more extensive

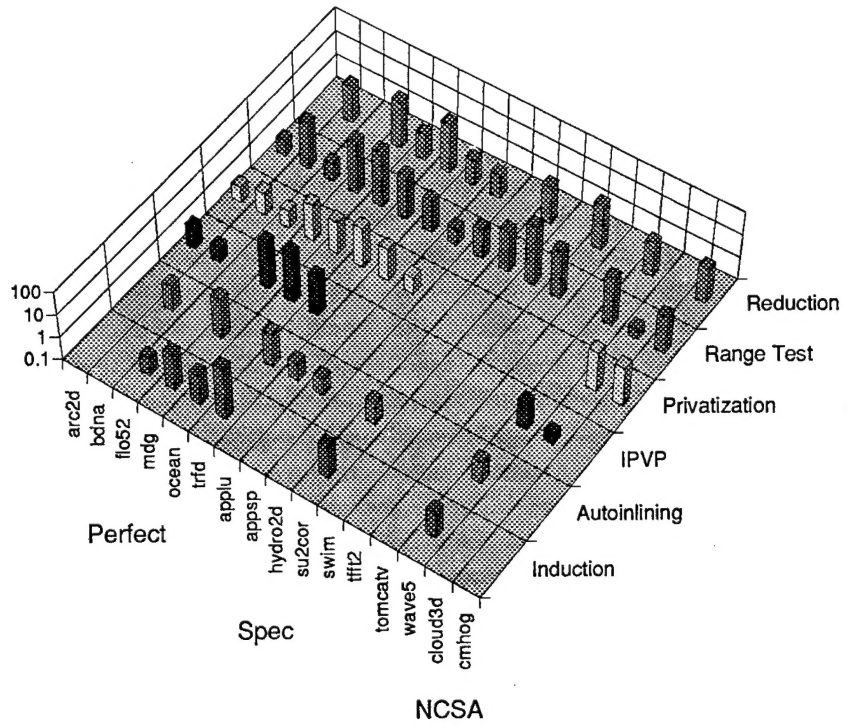

Figure 2: Percentage of Loops Serialized when Disabling Certain Techniques

experiments in the near future.

Third, Polaris needs to take into account additional program patterns, such as more complicated forms for induction and reduction variables, associative recurrences, multiple exit loops, and loops containing I/O statements. We have identified many important program patterns by analyzing many codes by hand. We plan to continue these hand studies and extend them to include collections other than the three mentioned above.

Fourth, we have to develop machine-specific optimizations to deal with important issues such as locality and communication. We are also developing machine-specific optimizations for shared-memory workstations [10]. Also, we have recently completed the first version of a code generator for the Cray T3D [18] and are now working on a similar module for the Convex Exemplar. The preliminary results on the Cray T3D are quite promising. These results indicate that in many cases it is possible to translate conventional code quite effectively for scalable machines.

We also must improve the efficiency of Polaris in order to allow us to compile very large Fortran programs. Although Polaris can routinely compile programs with 10,000 lines, we hope to eventually be able to compile programs with 100,000 to 1,000,000 lines.

6 Impact of the Polaris Project

The Polaris project has had a wide impact on both academia and industry. Group members have presented papers at a wide variety of conferences and workshops, winning a Best Paper Award

at the ACM International Conference on Supercomputing in 1995. The group bibliography contains more than 30 publications, including theses.

The group has produced seven graduates, 3 PhDs and 4 Masters. Some of these students are now employed by top computer companies (Intel, Hewlett-Packard, Silicon Graphics, and Microsoft). and one is a faculty member at Texas A&M University.

Polaris technology has generated much interest from several industrial compiler groups. We have maintained close working ties with Kuck and Associates, and Polaris technology is being integrated into their KAP parallelizer. Interest within IBM has culminated in a partnership award that will allow us to target Polaris at the new COMA multiprocessor under development at T.J Watson Research Center. Interest at SGI, Convex, and Sun has generated much interaction between their compiler groups and ourselves.

The Polaris parallelizer has been well-accepted around the world. We now have 24 licensed Polaris users. Twelve are from the US. The other twelve are from Canada, Europe, Japan, Korea, and Australia. Eleven users have been gained in the last six months.

These users are using Polaris in a variety of ways. At Cornell University, Polaris has served as the core parallelizer in a compiler course, upon which students implement compiler projects. Polaris was also used in a graduate course at Toronto and will be used next semester in a similar course at Texas A&M University. At the University of Malaga in Spain, researchers have been testing the usefulness of Polaris for parallelizing sparse codes. In Barcelona, Spain, researchers are analyzing codes for instruction-level parallelism with Polaris, using the powerful analytical capabilities of Polaris to inform the code generation module, which they have written. The University of Toronto is using Polaris as the basis for their NUMAchine parallelizer. At Purdue, they are targeting Polaris at a Sun multiprocessor workstation. At the University of Rochester, Polaris is being used as an integral part of a data access visualization environment (DAVE).

Several of the most active users will join us in a POLARIS exhibit at Supercomputing '96 in November. We will give on-line demonstrations of the variety of uses to which Polaris is being put.

7 Conclusions

The techniques implemented in the first version of Polaris have proven quite effective on a variety of programs. This verifies the results of our study involving the Alliant machine. Even so, we find that further progress is necessary. A wide range of techniques is necessary to detect as much parallelism as is present in real programs.

Perhaps the most important achievement of our work in the Polaris project has been the demonstration that substantial progress in compiling conventional languages is possible. There have been, and still are, many who do not believe it is possible to develop compilers capable of generating effective parallel code for a wide range of real programs. We disagree, and believe that, at least for Fortran and other similar languages such as MATLAB, effective parallelizers will be available within the next decade. Our experimental results and careful hand analysis of real codes strongly support this opinion.

References

[1] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, William Pottenger, Lawrence Rauchwerger, and Peng Tu. *Advanced Program Restructuring for High-Performance Computers with Polaris.* Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev. CSRD Technical Report 1393. To appear in *IEEE Computer.* December 1996.

[2] William Blume and Rudolf Eigenmann. Symbolic Range Propagation. *Proceedings of the 9th International Parallel Processing Symposium*, April 1995.

[3] William Blume and Rudolf Eigenmann. An Overview of Symbolic Analysis Techniques Needed for the Effective Parallelization of the Perfect Benchmarks. *Proceedings of the 1994 International Conference on Parallel Processing*, pages II233 – II238, August, 1994.

[4] William Blume and Rudolf Eigenmann. Demand-driven, Symbolic Range Propagation. In C.-H. Huang, P. Sdayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing. 8th International Workshop. Columbus, OH, USA. Lecture Notes in Computer Science 892*, pages 141–160. Springer-Verlag, August 1995.

[5] William Blume and Rudolf Eigenmann. Non-Linear and Symbolic Data Dependence Testing. Technical Report 1490, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., May 1996. Submitted for publication to the *IEEE Transaction on Parallel and Distributed Systems.*

[6] William Blume and Rudolf Eigenmann. The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. *Proceedings of Supercomputing '94, Washington D.C.*, pages 528–537, November 1994.

[7] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: Improving the Effectiveness of Parallelizing Compilers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing. 7th International Workshop. Ithaca, NY, USA. Lecture Notes in Computer Science 892, Springer-Verlag*, pages 141–154, August 1994.

[8] William Blume, Rudolf Eigenmann, Jay Hoeflinger, David Padua, Paul Petersen, Lawrence Rauchwerger, and Peng Tu. Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing. *IEEE Parallel and Distributed Technology*, 2(3):37–47, Fall 1994.

[9] William Joseph Blume. *Symbolic Analysis Techniques for Effective Automatic Parallelization.* PhD thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., June 1995. CSRD Technical Report 1433.

[10] Rudof Eigenmann, Insung Park, and Michael J. Voss. Are parallel workstations the right target for parallelizing compilers? In *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing, San Jose, California*, August 8-10, 1996.

[11] Rudolf Eigenmann, Jay Hoeflinger, and David Padua. On the Automatic Parallelization of the Perfect Benchmarks. Technical Report 1392, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., December 1994. Submitted for publication to the *IEEE Transaction on Parallel and Distributed Systems.*

[12] Keith A. Faigin, Jay P. Hoeflinger, David A. Padua, Paul M. Petersen, and Stephen A. Weatherford. The Polaris Internal Representation. *International Journal of Parallel Programming*, 22(5):553–586, October 1994.

[13] Keith Aaron Faigin. The Polaris Internal Representation. Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., 1994. CSRD Technical Report 1447.

[14] John Robert Grout. Inline Expansion for the Polaris Research Compiler. Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., May 1995. CSRD Technical Report 1431.

[15] Jee Ku. The Design of and Efficient and Portable Interface Between a Parallelizinbg Compiler and Its Target Machine. Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., May 1995. CSRD Technical Report 1500.

[16] Thomas Lawrence. Implementation of Run Time Techniques in the Polaris Fortran Restructurer. Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., May 1996. CSRD Technical Report 1501.

[17] David A. Padua, Rudolf Eigenmann, and Jay P. Hoeflinger. Automatic Program Restructuring for Parallel Computing and the Polaris Fortran Translator. *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, San Francisco, CA, February 15-17, 1995*, pages 647–649, February 1995.

[18] Yunheung Paek and David Padua. Automatic parallelization of noncoherent cache multiprocesssors. In *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing, San Jose, California*, August 8-10, 1996.

[19] Bill Pottenger and Rudolf Eigenmann. Parallelization in the Presence of Generalized Induction and Reduction Variables. Technical Report 1396, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., January 1995.

[20] Bill Pottenger and Rudolf Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. *Proceedings of the 9th ACM International Conference on Supercomputing, Barcelona, Spain*, July 1995.

[21] William Morton Pottenger. Induction Variable Substitution and Reduction Recognition in the Polaris Parallelizing Compiler. Master's thesis, Univ of Illinois at Urbana-Champaign, Cntr for Supercomputing Res & Dev, December 1994. CSRD Technical Report 1393.

[22] Lawrence Rauchwerger. *Run-Time Parallelization: A Framework for Parallel Computation.* PhD thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., August 1995.

[23] Lawrence Rauchwerger, Nancy M. Amato, and David A. Padua. Run-Time Methods for Parallelizing Partially Parallel Loops. *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 137–146, July 1995.

[24] Lawrence Rauchwerger, Nancy M. Amato, and David A. Padua. A Scalable Method for Run-Time Loop Parallelization. Technical Report 1444, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., June 1995.

[25] Lawrence Rauchwerger and David Padua. Run-Time Methods for Parallelizing DO loops. In M. Mango Furnari, editor, *Massive Parallelism. Hardware, Software, and Applications*, pages 1–15. World Scientific, 1994.

[26] Lawrence Rauchwerger and David Padua. The PRIVATIZING DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization . *Proceedings of the 8th ACM International Conference on Supercomputing*, pages 33–43, July 1994.

[27] Lawrence Rauchwerger and David Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *Proceedings of the SIGPLAN'95 Conference on Programming Language Design and Implementation*, June 1995.

[28] Lawrence Rauchwerger and David Padua. Speculative Run-Time Parallelization of Loops. Technical Report 1339, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., March 1994.

[29] Lawrence Rauchwerger and David Padua. Parallelizing WHILE Loops for Multiprocessor Systems. Technical Report 1409, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., March 1995.

[30] Lawrence Rauchwerger and David Padua. The Privatizing DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization. Technical Report 1383, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., October 1994.

[31] Lawrence Rauchwerger and David A. Padua. Parallelizing WHILE Loops for Multiprocessor Systems. *Proceedings for the 9th International Parallel Processing Symposium*, April 1995.

[32] Peng Tu. *Automatic Array Privatization and Demand-Driven Symbolic Analysis.* PhD thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., May 1995. CSRD Technical Report 1432.

[33] Peng Tu and David Padua. Automatic Array Privatization. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing, Portland, OR. Lecture Notes in Computer Science.*, volume 768, pages 500–521, August 12-14, 1993.

[34] Peng Tu and David Padua. Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 414–423, July 1995.

[35] Peng Tu and David Padua. Efficient Building and Placing of Gating Functions. *Proceedings of the SIGPLAN'95 Conference on Programming Language Design and Implementation*, June 1995.

[36] Stephen Weatherford. High-Level Pattern-Matching Extensions to C++ for Fortran Program Manipulation in Polaris. Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dév., May 1994. CSRD Technical Report 1350.